Haaga-Helia
University of Applied Sciences

Automating mobile application deployment using a CI platform in small-sized enterprise

Mikael Mäkelä

Bachelor's Thesis
Degree Programme in Business Information Technology
2018

Abstract

Haaga-Helia

16 May 2018

Author(s)
Mikael Pauli Olavi Mäkelä

Degree programme
Business Information Technology

Report/thesis title
Automating mobile application deployment using a CI platform in small-sized enterprise

Number of pages and appendix pages
31

This project thesis is about implementing a Visual Studio App Center continuous integration platform. The goal of integration is to aid the commissioning party to develop and deploy mobile applications to the users and testers. To solve the problem of the commissioning party, it is necessary to configure the build and distribution processes.

Continuous integration(CI) is a software engineering practice that allows developers to avoid integration problems by constantly merging code changes to the repository. Using a CI platform, every time a merge occurs the repository is build and the code is tested. Constant building and testing help the developer to see if there was a problem with the new code and where it occurred.

Continuous integration relies on version control. Version control is a system that records changes to directories and files that it contains. Using version control, developers can submit changes from different clients using branching and merging.

Visual Studio App Center is a CI platform that is created for mobile applications and supports all native applications and most cross-platform frameworks. App Center can build code from Git repositories, run UI tests on successful builds, and distribute them via email or to distribution services. Additionally, App Center provides tools to keep track of builds using webhooks and status badges and to create issues in repository service using the bug tracker.

Keywords
Continuous Integration, Visual Studio App Center, Version Control, Mobile development

Table of contents

1 Introduction 1
1.1 Presentation of commissioning company – Kehätieto Oy 1
1.2 Presenting problem domain 1
1.3 The objectives 2
1.4 Scope 3
1.5 Process description 4
2 Theoretical Background 5
2.1 Version Control and Git 5
2.2 Continuous Integration 6
2.3 App Center 6
2.4 Structure of the mobile applications 8
3 Building process 10
3.1 Objectives 10
3.2 Implementation and results 10
3.2.1 Version control 14
3.2.2 Build configuration 14
3.2.3 Build scripts 16
4 Distribution process 19
4.1 Objectives 19
4.2 Implementation 19
4.2.1 Android 20
4.2.2 iOS 23
4.3 Results 25
5 Additional CI configurations 27
5.1 Bug tracker 27
5.2 UI tests 28
5.3 Webhooks 28
5.4 Build status badge 29
6 Conclusion 31
6.1 Future of App Center in Kehätieto Oy 31
References 32

Table of Figures

Figure 1. Build menu with list of branches....12
Figure 2. Single branch build log....13
Figure 3. Build configurations....15
Figure 4. Android build signing in Build configuration menu....21
Figure 5. Build distribution in Build configuration menu....21
Figure 6. Create service account menu in google developers console....22
Figure 7. iOS build signing in Build configuration menu....24
Figure 8. Bug tracker configuration menu....27
Figure 9. Toggle for launch test in Build Configuration menu....28
Figure 10. Message from App Center in Slack....29
Figure 11. Build status badge in Build configuration menu....29
Figure 12. Status badge table from documentation....30

Terminology and Abbreviations

Bash	Unix command language
Branch	A snapshot of the codebase from which the branch was created
Digital distribution service	A service in which the mobile applications, among other multimedia, can be down-loaded.
DLL	Dynamic Linked Library
Git	Open source version control system.
HTTP post	A web API response which sends infor-mation
IDE	Integrated development environment.
JSON payload	Information sent through web API in JSON format
Origin	remote repository
Root	Original branch in Git version control
TFS	Team Foundation Services
Version Control	A system that has directories and files, which it tracks
VSTS	Visual Studio Team Services
Webhook	An HTTP post with a trigger
Xamarin framework	.NET cross-platform framework
Xamarin.Forms	Cross-platform UI toolkit

1 Introduction

1.1 Presentation of commissioning company – Kehätieto Oy

Kehätieto Oy is a small enterprise which is specialized in web application development using ASP.NET web framework that utilizes C# or VB.NET for business logic and HTML, CSS and JS for front-end, and SQL server for the database. Kehätieto has been operating since 1989 but started working on software development since 1997. In Kehätieto, the primary operations are web application projects and secondary operations are front-end development and tech support. Under web application development department there is also a team that does mobile applications. The mobile team is the most recent addition to the company, but due to the clients demand of mobile applications, they are expanding the mobile team and looking for more staff to work on it.

1.2 Presenting problem domain

Currently, in Kehätieto the mobile team is understaffed and is having difficulties with the development cycle, which can be complicated and time-consuming. The main problem is deployment and distribution. In web application development the distribution is simplified because websites are running on company's servers and changes are easy to make by replacing old files with new ones. However, in mobile development, the only way to publish the application is by creating a binary package, which is a pre-built executable code. (goldilocks 8 May 2014.) Because of this, it is not possible to replace one page in the mobile application, rather a whole package must be replaced in a digital distribution service. This creates the need to publish only working versions to reduce the possible downtime or buggy release. Other key difference in deployment, is that mobile applications require different certificates to successfully deploy packages to application stores. Managing those certificates can be time-consuming. Because of the time and resources that go into deployment, the company decided to automate it and integrate Continuous Integration(CI) tools. Previously, the company was using HockeyApp for installation of new versions of apps, but that was the only feature that was used from HockeyApp.

Currently, Kehätieto is maintaining nine applications in stores, three of which are in active development and one is in the planning stage. All those applications are at least on iOS and Android platforms and most of the maintained ones are supported on all three mobile platforms. This adds up to a big amount of applications that need maintenance and testing. With a small team of developers and a big number of projects, using time and re-

sources efficiently was a priority, thus some automatization was required. Because Kehätieto is mostly using Microsoft technology stack, it was decided to take the Visual Studio App Center as CI tool, which recently was released from preview version.

When the project was started, there was a difficulty with the way the mobile applications were built in Kehätieto. To save resources and time, Kehätieto created a mobile application project template, which had base implementations of the applications. Project template was then imported into every other application as a dynamic-link library(DLL) and the projects used it as a resource. To understand the problem, it is important to understand what a dynamic-link library is. As stated on Microsoft Dev Center web page (Microsoft.) “DLL is a module that contains functions and data that can be used by another module (application or DLL)”. DLL is a collection of code that cannot be executed on its own, compared to EXE module. DLL modules can be imported to a solution with other DLL modules or EXE module and accessed using public or external methods. DLL provides the benefit of modularity, where one library can be imported into many projects. However, the disadvantage of DLL is that once it is imported to a solution, it is already compiled, which means it cannot be debugged. This solution was working, when the number of applications was small, however, it was a short-term solution, because the more there were projects that used template project as DLL, the more maintenance the template project was requiring. At some point, it became impossible to support DLL option, because the main projects were crashing because of the DLL and there was no way to debug it.

Additionally, during the analysis, previous mobile team lead developer gave his opinion about the publishing process of the application to the digital distribution services. He expressed concern towards the certificates and how challenging it was to maintain them. Apple’s publishing process could have taken a long time due to the Apple checking the application before allowing them on the App Store and all the certificates always needed to be active. Google publishing process, while less complex, could have also taken its time because everything needed to be done manually. All those certificates needed to be stored locally and a regular backup was needed to be done, so that publishing process would not have been compromised. (Ollari 22 March 2018.)

1.3 The objectives

From the analysis of the problem, it is possible to derive the objectives that need to be fulfilled during this project. The main objective and the final deliverable is a functional deployment pipeline to simplify the development cycle in development and maintenance

stage. There are several targets that need to be hit during the project. At the start of the project, it is necessary to test App Center functionalities first with a test project and learn how App Center works. At this stage, it is also required to research how linking repositories to App Center works, how to distribute release builds and what is the best way to do it, how to implement version control and how to upload finished products to stores. After the initial preparations, mobile team will start changing the existing Visual Studio Team Service repositories into Git repositories and after that, it will be possible to link projects to App Center.

The tangible result of this project is the updated version control with one solution containing all mobile application projects and App Center platform that will be integrated into to Kehätieto Oy for mobile development. Because there will be done some adjustments and improvements to workflow with App Center, the time used to distribute the application will be shorter, allowing the developers to concentrate more on coding the applications.

During the project, the aim is to learn how the software deployment pipeline is created and managed. That also implies learning and researching topics that will be done while implementing it. The topics that will be covered, including but not limited by, are Git, continuous integration, publishing application and Bash and PowerShell commands because App Center uses those scripting languages for build scripts.

1.4 Scope

The task is to implement the App Center distribution and version control mechanics, and to lay the groundwork for further implementation of the rest of App Center features like testing, collecting and analyzing data, and sending a push notification to user groups.

During the project, the functionalities of App Center that do not have a direct correlation to the deployment pipeline won’t be focused on, which means that collection and analyzation of the user data and push notifications won’t be covered at all. UI tests, however, will be covered, but only briefly.

Additionally, there will be no coverage of the Windows Phone implementations. This is because App Center does not support the Windows Phone version that is currently being developed in Kehätieto, however, there are more reasons for this decision, which will be covered later in thesis.

1.5 Process description

In this thesis, the approach to the writing will be to first learn how to perform a certain task by reading documentation and testing. After understanding how to do the task, the findings will be documented and then finally the features will be implemented, if they weren't applied during testing. After the implementation, the documentation will be completed with the result and what does that feature do.

During writing, the focus will be only on one project which will have apps for Android and iOS platforms. This project is for the Finnish Medical Society Duodecim. While the configuration will be done to all the other apps as well, for the documentation purposes Duodecim projects will be used as examples.

2 Theoretical Background

Before the practical implementation of the project could be initiated, the research took place. During research there were topics and aspects of the projects, which were highly technical and not accessible without a technical background. This chapter will be focused on explaining key topics of this project in a clear way as well as give a prelude to what will be done in this project.

2.1 Version Control and Git

When developing software, developers need to be able to save the changes that have been done to the code and see what changes were applied to code base previously. To help developers with those tasks, they can use version control, which is a system that tracks and saves the changes to the code. When a code is uploaded to version control, it applies the changes and creates a log of what the code has changed. (Collins-Sussman & Fitzpatrick & Pilato 2011.) According to Git terminology, a piece of code that is uploaded is referred to as a commit and uploading process is called a push. There can be multiple commits during one push because commits save changes to a local repository, while the push is saving changes from the local repository to remote repository. (Tower.) Repository, in turn, is a place where the code is being kept and whenever a change is made to files in it, it keeps all the versions of it. A repository can be local or remote. In Git, the system can give the remote repository an alias *origin*. (Collins-Sussman & Fitzpatrick & Pilato 2011.)

An essential feature of the version control is being able to create branches. When a repository is initiated, by default it has one branch which is a *root*. In Git by default, it is called *master* branch. Any new branch that is created is either a copy of root or a copy of another branch. Branching can be helpful when developing new features without interfering with other developers or pushing bad code to master. With branches, it is possible to always keep master branch functional, so that all new branches from master would always be functional as well. When a feature has been done, it is possible to merge the code back to the root. Merging is a complicated process that analyses both branches and attempts to automatically merge files and lines of code between the two. For best result, it is advisable to first merge the master from origin to local branch, in case someone had made any changes to it and then if the merge was successful, push the code back to root. Occasionally, however, merging doesn't succeed and causes a merge conflict, which requires the developer to solve the merge manually. (Nagele.)

Sometimes, it is necessary to control what code is pushed to the root. In these situations, it is possible to set the root to accept the code only in form of pull requests. Pull request is a version control mechanic that allows code to go into a review before being merged to the root. When a pull request is opened, other developers can review the code and discuss it in the version control service. Pull request is a way to stop a bad code from going into the root. (GitHub.)

2.2 Continuous Integration

To solve the problem that Kehätieto is having with the mobile development, there is a need for continuous integration tool. Continuous integration (CI) is a software development practice where developers constantly merge their changes to the central repository. Merging the code can occur daily or several times a day and after every merge, the repository is built and tests are run against it. (Amazon Web Services(AWS).) The idea of CI is to avoid integration problems, which can occur when developers don't merge their code changes to the main repository often enough. This can lead to different merge conflicts, long integrations and in case the build will fail, it will be harder to figure out what exactly is failing. CI allows developers to quickly find errors and perform necessary actions to revert the broken merge or write a fix for it. (ThoughtWorks.)

To enforce the practice of the continuous integration, it is necessary to have a tool or a service that does the automated builds and testing, and a certain culture where developers merge their changes often. While culture is something that can be changed, it was necessary to find a suitable CI or build service. Currently, on market, there are many different options to choose from, ranging from open source build services to enterprise level CI platforms. However, because the main priority was that the CI service has a support for mobile applications built on Xamarin, it narrowed the search down to two options, Visual Studio App Center and Appveyor. However, the decision was made in favor of App Center because it suited the needs of the mobile team more, since it App Center is more targeted for mobile application, while Appveyor is generally used as a CI tool for .NET application with support for Xamarin.

2.3 App Center

Visual Studio App Center is a continuous integration tool created by Microsoft corporation. The preview version, which was called Visual Studio Mobile Center, was announced on 16th of Novem-ber 2016 on Microsoft Connect(); 2016 conference with several key features, such as building, testing, distributing and monitoring the apps for bugs, crashes,

and other analytics. (Friedman 2016.) On Microsoft Connect(); 2017, on 15th of November 2017 it was released out of preview and was renamed to Visual Studio App Center. (Ballinger 2017.) After the release, App Center included all the same key features as the preview had, but with additional functionalities.

As stated before, App Center has features such as building repositories, running tests against successful builds, distributing those builds and collecting usage data. Because those features are essential to this project, is it important to explain how those features work and during which order they are executed.

When a project is created in App Center, the user must link the repository and App Center. During the linking, a service webhook is created in the repository service. A webhook is a web callback that occurs when something happens. (Quinlan 2014.) For every webhook that is created there needs to be a trigger which activates it. In case of App Center, the webhook is triggered when a repository senses that new code was pushed. When the webhook is activated, it sends an HTTP post containing JSON payload, which is the information about the event that was triggered. The technology that enables this kind of communication is called Web API.

When App Center receives the post from the repository, it activates a clean virtual machine in their data center and begins the build process. (Pasat 2017.) Build process begins with virtual machine downloading necessary software for compilation which depends on the platform that the code is intended for and then using the HTTP Post that it received, downloads files from the repository. The build process continues by the system acquiring necessary NuGet packages and compiling the code into a binary. This can be seen in the build output in App Center.

The test process is initiated after the build was successful. The test that App Center supports are graphical user interface tests or simply UI tests. UI tests are not unit tests that test the code, rather UI tests that test if the application works correctly using different UI elements like labels, buttons etc. (Ghildiyal & Chandra & Guru99.).

After building and testing has been done, and builds were successful, it is possible to distribute the apps to the users or testers. This is can be done via an email notification that contains the link to the newest version or it is also possible to upload newest versions to Google Play service, iTunes Connect or Intune Company Portal. It is also possible to in-

clude a NuGet package that is provided by App Center that provides a possibility to update the application on start-up like the link form the email notification. Before App Center, Kehätieto Oy used these kinds of updates using HockeyApp.

Lastly, in App Center, it is possible to collect statistics about the usage of the application, crash reports, events, and errors. App Center can also collect information to create audiences to whom push notifications could be sent.

2.4 Structure of the mobile applications

As stated previously, it was necessary to find a CI platform that would support Xamarin mobile development framework. Supports for the Xamarin framework was a priority because all the applications that have been built at Kehätieto Oy are using it. This decision was made when the mobile team was formed and the main reason for this was that all business logic could be shared between all mobile platforms. Since the UI of the applications was planned to be simple, it was decided to use Xamarin.Forms, which provides cross-platform UI toolkit for .NET developers.

Xamarin.Forms was introduced on 28th of May 2014. The main idea behind Xamarin.Forms is to create a level of abstraction which maps different native elements during compilation of the software. The reason why Xamarin.Forms is useful is because it is required to write the UI code only once using XAML, Microsoft own markup language, to create a shared UI across all platforms. However, this is also a problem because Xamarin.Forms can access only elements that exist on all platforms, meaning it cannot do any platform specific elements. (Stone 2018; Petzold 2016, 15.)

Creating applications using Xamarin.Forms, enforces a couple of architectural patterns to be used, to get the most out of the framework. Because Xamarin prefers business logic separately from platform specific code, it is advised to use either Portable Class Library (PCL) or Shared Asset Project (SAP). Both serve a similar function; however, they have a difference in how they behave. When a project is built with PCL, it gets compiled into a DLL, which gets linked to applications at runtime. SAP however during compilation gets dissolved into each application as if they are one, effectively creating 5 copies of itself to each application. Another architectural pattern that is recommended to use the is the MVVM architectural pattern, which allows creating bindings in XAML from the view to the underlying view model and model. (Petzold 2016, 29; 418.)

In addition to those standard architectural patterns, in Kehätieto Oy, it was decided to create a template project for all mobile apps that would be linked to a consecutive project. As discussed before, this pattern caused issues and required reworking.

3 Building process

This chapter will cover steps that were done to implement building functionality from App Center, which also includes tasks that were done to allow the App Center integration.

3.1 Objectives

When the mobile team decided to start the improving the current troubled development cycle, priority was the architecture of the applications and the version control approach. As it was established in the theoretical background, the way the applications were previously structured was causing errors that were hard to debug. The first step was to put all the projects under one solution and create a git repository which would contain it. Putting projects to Git repository was a unanimous decision due to the App Center integration, however unifying all the projects under one solution was heavily debated. This was suggested because this would make the debugging easier since the template application wasn't a DLL anymore. However, the problem with this was that the size of the solution would grow every time a new project would be started and managing it would become a major concern over time.

Next step was to link the repository containing the code to App Center. This was done after the repository was created. Because each project was added separately, the linking process happened every time a new project was added to the repository. To build the projects in App Center, each build was necessary to configure. This required setting necessary configurations, adding certificates for release versions and writing possible global variables. The process of the linking will be described in detail in the implementation section.

After the build configuration was completed, it was required to create build scripts that would rename the application according to the configuration they are in.

The result of finalizing this part should be working building process. The output of this process needs to be two binaries of each application, one of which is a test version and other is a release version, which could be distributed.

3.2 Implementation and results

When the App Center integration initiated, it was decided that first step would be to migrate all the current projects from TFS repositories to Git repositories. Because migration to the different repository would mean that all the unfinished code would be discarded, it

was decided that all the code which developers were working on at that time should first be checked in, meaning uploaded, to the repository. While the developers were finishing up their changes to the code, it was suggested that all the mobile projects would be unified under one repository due to the debugging issues that were covered earlier. As this idea was accepted, the need to migrate all the projects to Git was removed. The new requirement was to create one new Git repository and then start adding the projects one by one to the main solution. Because adding the projects was not directly part of App Center integration and required in-depth knowledge of the code, one of the senior developers was tasked to do this instead.

The work in App Center starts with creating the projects. First, it is necessary to add the project by pressing “Add new app” button on the main screen and then give a name to the application, write an optional description, and choose OS and platform. Main platforms supported by App Center are iOS, Android, Windows, and macOS which is in the preview. Then in Platforms, there is Objective-C / Swift, Java or Unified Windows Platform (UWP) for iPhone, Android and Windows respectively as a native development platform and then there are cross-platform frameworks Reach Native, Xamarin, and Cordova in the preview. Since all Kehätieto’s projects are done in Xamarin, all that needs to be specified is which OS the project is meant for. In this case, it will be iOS and Android.

The reason why Windows projects aren’t created is that Kehätieto’s Windows apps are built for the Windows Phone 8 and 8.1. Because only Windows 10 mobile supports UWP projects, the applications created for previous versions of Windows Phone can’t be upgraded. Since App Center supports only UWP mobile applications, it is not possible to add Window Phone projects from Kehätieto.

For clarity and consistency, mobile team decided that the projects in App Center should have a naming convention. The convention that was agreed on dictates that a project must be named after the applications name and the platform that the project is meant for separated by a dot. An example of this would be *Duodecim.Android* or *Duodecim.iOS*.

After the project is created and selected, a dashboard appears. In this dashboard, a user can configure everything App Center has to offer. The first thing that needs to be done is select the build tab. There the user is asked to connect to the service that is hosting the project. The choices that App Center currently supports is Visual Studio Team Services, GitHub, and Bitbucket. It is important to notice, while it is possible to connect to any of those three repository services, the repository that has the code must support Git version control.

When the App Center and the repository are linked together, App Center gets information about all the branches that have been created in that repository. On figure 1 can be seen how the build menu looks like when the repository has been connected. It is possible to see all the branches that exist in the repository and which have been built, failed to build or not built at all. It is also possible to see when the branch was last built, which it the last commit and if there is a trigger for the build. The list is organized in alphabetical order with built branches at the top.

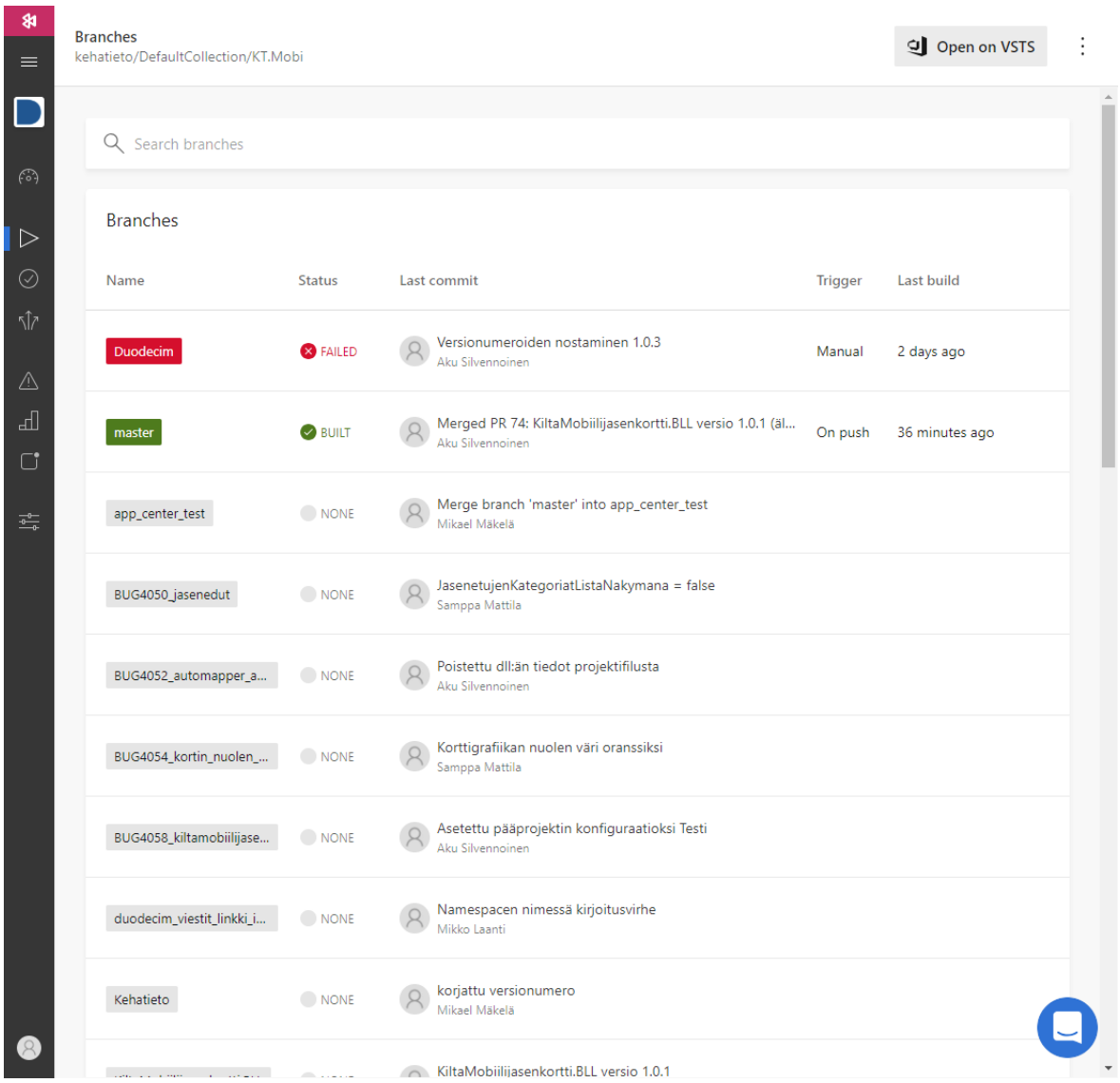

Figure 1. Build menu with list of branches

When one of the branches is selected, a new tab opens with a log, which can be seen in the figure 2. In this tab, it is possible to see when the branch was built, during which commit that build happened, and if it succeeded or not. When one of the builds is selected a new tab opens that contains the full build output. In this output, it can be possible to see if

the build had any errors and where those can be found, which tests and build scripts were ran and if they succeeded. When App Center builds the code, sometimes it picks up errors which the development environment doesn't and results in a failed build. Reason for a failed build can be a failed build script or UI test, or something wrong with the code, like missing a reference or generally broken code. (Adams 2017.)

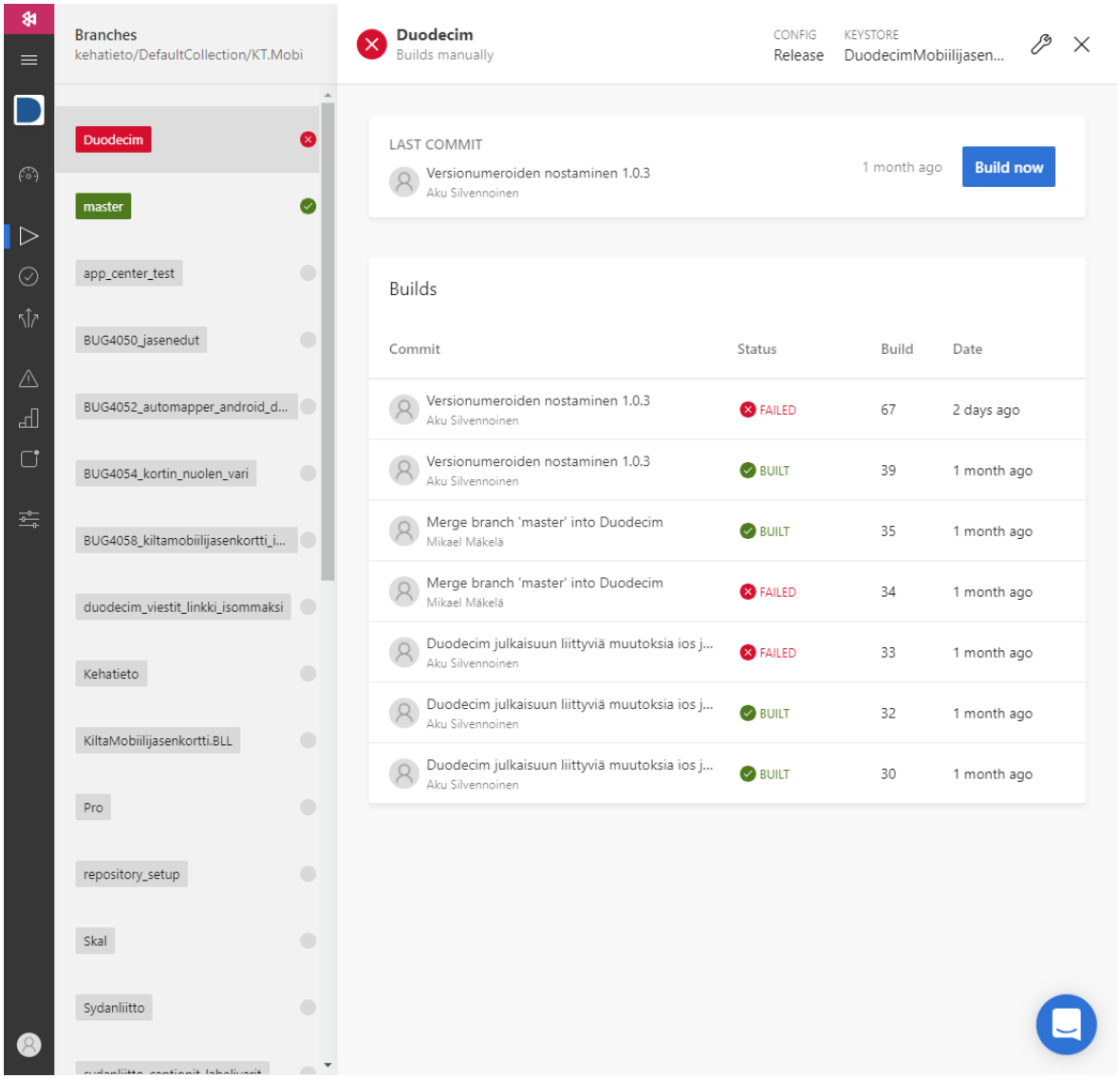

Figure 2. Single branch build log

In branch log tab the user can click on the tool logo and then opens the tab that has the configuration of the branch. If the branch hasn't been configured yet, the button to configure the branch will be visible instead of the build log.

3.2.1 Version control

After the repository in VSTS was configured, it was required for the mobile team to decide on the version control issues. Right away it was decided that master branch should always be functional and any code that will be merged to it should be merged via pull request. This way it would be ensured that no broken branches would be created from master and test version could be created from master at any moment.

To enforce master always being functional, it was decided that whenever a new feature or bug fix was worked on, it would be done in a separate branch, with an appropriate name that would describe what the branch contains. All the bug fix branches would start with BUG and number of the issue in VSTS and new features would start with US and the number of the issue. As an example, a new feature would be named US4123_updated_navigation.

It was decided that the release branches would be named after the project name, which can be seen in figure 2. Only a senior developer on the mobile team can alter those branches. The way release branches are updated is by cherry-picking relevant commits and merging them to release.

3.2.2 Build configuration

As seen in figure 3, the build configuration tab has several settings that control how the branch behaves in App Center. The first step is to pick the .csproj file and set it to the project that needs to be built on this build. Since the App Center project that has this branch is called Duodecim.Android, it points to that this build needs to set Duodecim.Droid.csproj file as the project. Next step is to set the project configuration. Normally, it is either debug or release, but developers can configure their own configurations based on their needs. Because this build is not meant to be published, the configuration will be set to debug.

Next, depending on the platform, there are either one or two choices to make. On Android, the user needs to select which version of Mono should the application use. Mono is a framework based on Microsoft's .NET framework, which supports all windows programs. Mono is an open source implementation of .NET Framework, which is designed to easily create cross-platform applications. (Mono.) When picking which version should the application use, it is best to always take the newest one that App Center supports, because in case the project contains features from a new version of Xamarin, the older Mono versions might not support it. However, choosing an older version might allow the support of

older applications. (Pasat & Chew 2018.) By default, App Center suggests Mono be the newest and marks it with a Stable-tag. For iOS App Center also requires choosing the Xcode version, which will be used to run the build on. When the project is started user has a possibility to choose which version of Xcode would be used for building, but as of the moment of writing, only current version, which version 9.3, is available.

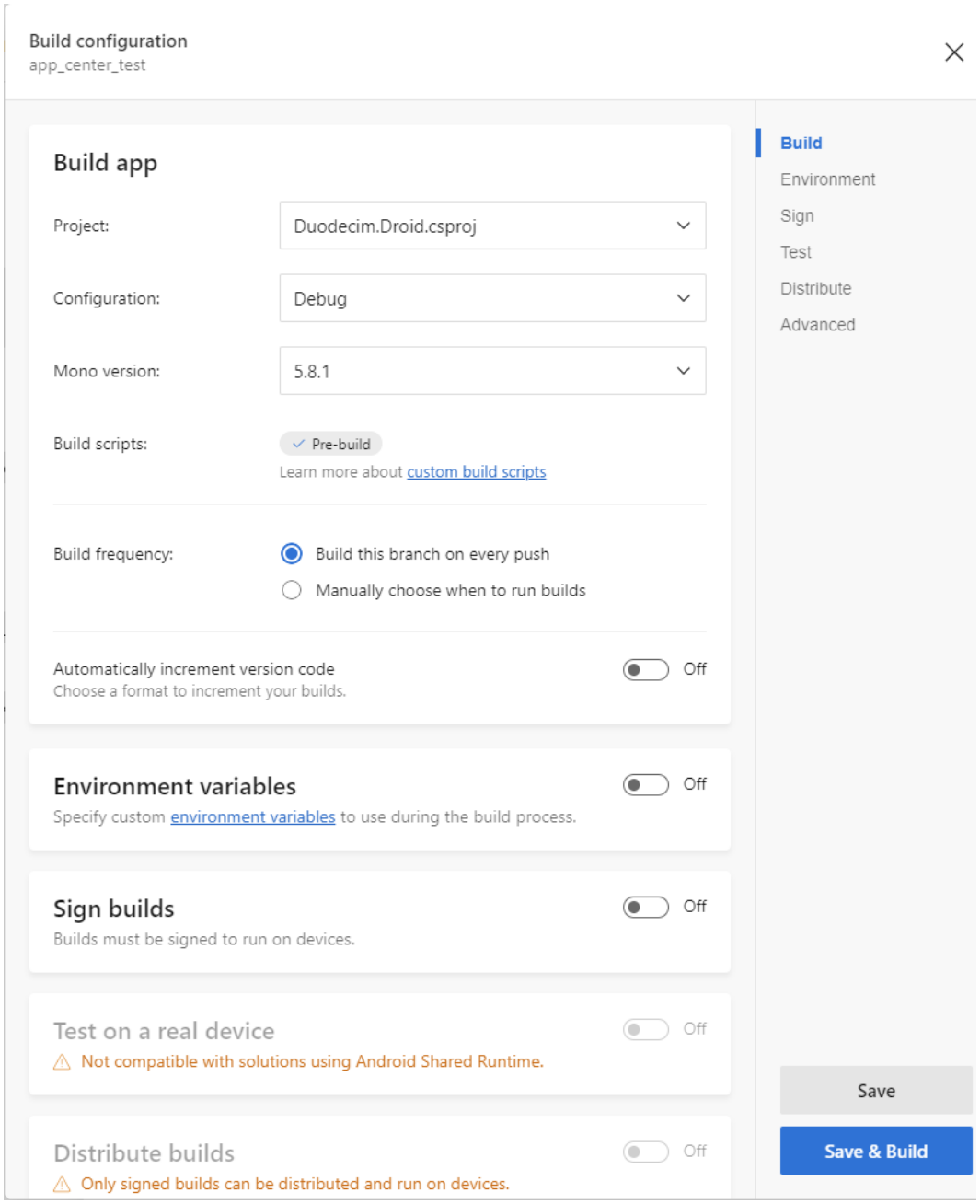

Figure 3. Build configurations

Under Xcode and Mono version picker in figure 3, there is a field which has a checkmark next to “pre-build” text. This signifies that the repository contains a build script. Build scripts will be covered in detail in the chapter 3.2.3 Build scripts.

After configuration for the code is done, a user can choose how often the build is built. It can be built every time code is pushed to the repository or it can be set to build only manually. Additionally, it is possible to set build number that would increment every time the build is built. The version number can be a build ID, which is a unique number that is assigned to build, or it can be a Unix timestamp in seconds.

It is also possible to set custom environment variables for the whole application to use during compilation of the build. Those variables can contain non-sensitive data such as paths or values that can change easily, or sensitive information like passwords or app secrets. Environment variables can be accessed by build scripts or by the applications by referencing the variable key surrounded by percentile symbol in XAML or XML or by using dollar symbol and the variable key in build scripts. (Adams 2017.)

As an additional configuration for the build, App Center offers to sign the builds, test on real devices, distribute the builds, and activate build status badge. However, those will be covered in depth in next chapters.

Due to the mobile team deciding that it required two main branches, one development, and the other production, it was required to give them different configurations. Reason for this was that builds from these repositories were meant for different purposes.

Because both builds should be releasable, both must be signed with proper certificates, however, the configuration setting on them can't be the same. While release should go to the end users and connect to the production database, development build should connect to test database. Because it is possible to control what database the applications use via configuration, it was decided that production version would have Release configuration setting and the development version would have a custom Test configuration setting.

Other difference is that development version should be built constantly to see if there are any problems. Conversely, the production version should be built only manually, however, when it is built, it should be distributed right away to the digital distribution services.

Everything else needs to be the same, to ensure that the build works properly.

3.2.3 Build scripts

In App Center, it is possible to write build scripts. The build scripts are written in Bash or PowerShell command languages and they are intended to run during the build process.

There are three types of build scripts that are supported. A project can have all three scripts because each script is executed during a separate moment. If the scripts are in the same directory as the .csproj project file and are named according to the step during which they need be executed, all of them will be executed.

First one that is executed is the post-clone script, which is run immediately after the repository has been cloned. At that stage, no compilation has been done, so it is possible to alter the code by adding resources, such as NuGet packages.

The second script is executed during the pre-build step. During this step, it is possible to alter the name of the application, change version number or alter other general properties of the application.

The last script is executed during the post-build step. At this point, the application has been built and can't be changed anymore. At this step, the application can be sent to other services or tested using UI tests.

For this project, it was necessary to create a pre-build script that would alter the name of the application so that it would include the build version at the end of the name. This script was meant to run if the build was made from master branch. The requirement was to be able to send the binaries from master branch for testing purposes to clients and allow them to easily see which version of the application they have.

In order for a script to run on App Center, it is required that the shebang of bash script is in *#!/usr/bin/env bash* format, where env stands for environment. (Chadwick 2017.) This kind of shebang points to the interpreter that App Center is using. The reason why this kind of shebang is important is that, without env modifier, the path to the interpreter is absolute, meaning that if there isn't interpreter at this path, it will not execute the script. However, env modifier looks for the default interpreter in the current environment and uses it. This makes the path more flexible, however, it creates a possible security flaw, where someone could create a false interpreter with the name bash and system could pick the false one. Because the code is being built on App Center virtual machines, which are destroyed after each session this security flaw is avoided and env grants that flexible path. (Pasat & Dmitry & Galunin & Adams 2018.)

In the documentation, there was an example of possible scripts for all their stages. The example script which was written for pre-build step was close to what was needed to be

done, so it was taken as a template. The example provided all the parts that were required by App Center to run them.

When writing the scripts, the goal was to create as much shared code as possible, so that each project in App Center wouldn't have a slightly different script. In the script, there is an if-statement, which same in every version. The way it was implemented was that all changing parts were defined in custom variables and some variables were taken from App Center. However, there was a problem with finding a path to property file inside the virtual machine. The problem was solved by using the App Center variable *APPCENTER_SOURCE_DIRECTORY* and appending it with the environment variable that has the rest of the path. This way it is still necessary to write the path in every project in App Center, but the build scripts are more generic.

The other problem was with writing the scripts. The example that documentation provided was only meant to alter iOS application and even then, the script required some modifications to suit the needs of the mobile team. The main problem was to access info.plist, which is an iPhone application property list file. To solve the problem, extensive research was conducted, however, there was no official documentation on how to read information from that file, so it was necessary to read what other developers had already done. After trial and error, the right method of reading the properties was found and the script was working.

When writing the script for Android applications, the same problem appeared. It was difficult to read or write properties from AndroidManifest.xml because there was no tool provided by Google. This problem was solved by analyzing code and finding the lines of code that needed to be changed and instead of just changing the value, the whole lines were replaced with the same line but with a different value.

4 Distribution process

When the build process was completed, next step was to implement the distribution process. Simplifying distribution was one of the main priorities for Kehätieto because that was the most time-consuming process for the previous mobile team. This chapter will address the signing of the apps, how to get the right certificates, and what kind of distribution methods are available through App Center.

4.1 Objectives

The main objective of distribution feature for the mobile team is that when a release version is successfully built, it would instantly send a new version to the digital distribution services. Additionally, for testing purposes, there should be added a possibility to send testing versions to clients for them to test the application before actual release is made.

The first thing that needs to be configured is the email with a link to test builds, which would be sent manually to testers before the final release. To fine tune the distribution process, the email distribution will be configured first. After the packages have been tested to work through email, the connection to the stores will need to be configured and packages sent to the respective distribution services.

The priority during this step is to learn how certificates work in Android and iOS, how publishing each application works normally, and how it needs to happen using App Center.

4.2 Implementation

When planning on how to release the application, it was decided that there was a need for two different packages, one for external and internal testing and other for end users. For this, it was necessary to have two builds done on App Center. After discussion with the mobile team, the decision was made to have a release branch which would be configured to upload binaries straight to the digital distribution services and from master branch it would be possible to get the test versions.

For App Center to be able to distribute a binary, the configuration of the build must be set correctly depending on the platform. With build configuration, it is also possible to set what architecture to build on and which projects will be included in the building of the solution. (Goldberger 2017.)

By default, Android projects have two configurations: debug and release, while the iOS project has four: debug, release, Ad-Hoc and AppStore. However, it is also possible to create own build configurations in an integrated development environment (IDE), such as Visual Studio.

The importance of creating separate versions for testing and production is that it is possible to create different configurations for different builds, using Preprocessor directives or Conditional attributes in code. (KeithS 2011.) Those code features allow defining which code can be run during debug and which can't, allowing for setting different database connection for release version and different for debug version.

Because the release and distribution processes on both platforms are different, the implementation of them will be discussed separately.

4.2.1 Android

When deploying a Xamarin.Android application in App Center it is necessary to set the build configuration to be release or if it's a custom build configuration, then it needs to be configured so that it doesn't use Shared Mono Runtime. While it is not necessary for distribution to disable shared mono runtime, it increases the size of the binary and prevents it from being tested in App Center. Physical devices rarely have shared mono runtime installed on them, thus making them fail to run. Since App Center emulates physical devices, shared mono runtime is not included in them as well.

After the build configuration has been set, App Center requires the application to be digitally signed. In Android development, there are two parts to signing the APK. The first one is the public key of a public/private key pair. A public key is a unique ID of the application that associates the APK with the private key. A public key is signed automatically to the APK during the signing process. (Google Developers.)

However, the private key is included in a keystore file. This file can contain one or more private keys and it needs to be generated in an IDE to receive the private key. When signing the APK, it is possible to either import the keystore to add a new private key in it or create new keystore. When creating new keystore, it is necessary to write down at least one piece of personal information, give a password to the keystore and then write an alias and password for the private key. (Google Developers.)

After the keystore has been created, as seen on figure 4, it is necessary to import the keystore to App Center in Build Configuration menu and enter the password for the keystore and write key alias and password. When done, the application needs to be built and if it succeeds it will be signed and distributable.

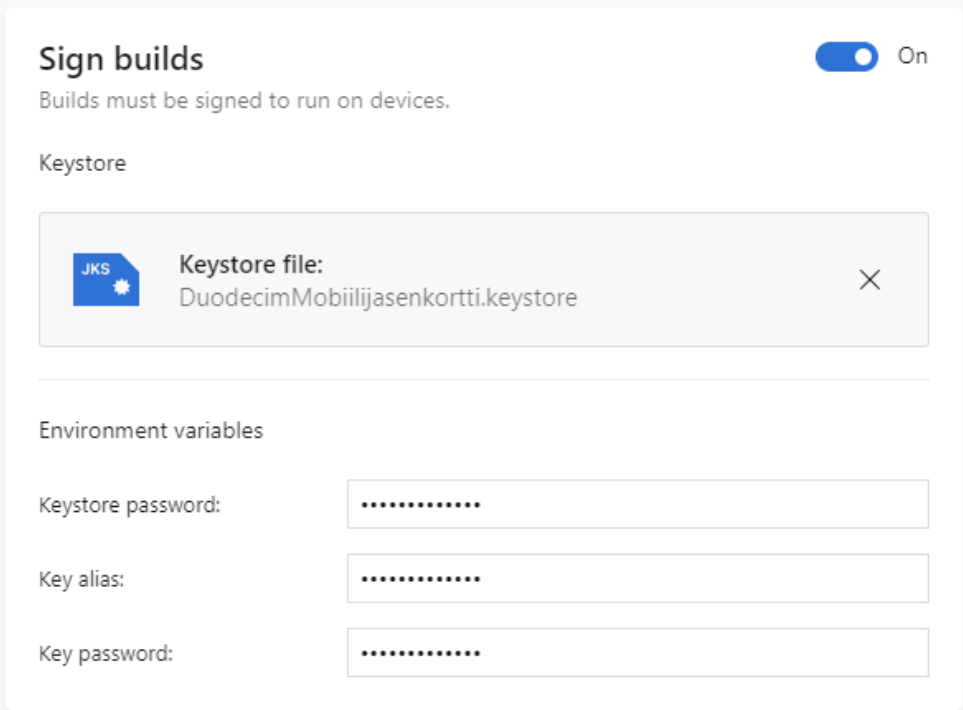

Figure 4. Android build signing in Build configuration menu

The distribution can be set in two places in App Center, one of which is in Build configuration menu, as seen in figure 5, and other is from the dashboard in distribute menu. If the distribution is done from Build configuration menu, it needs to be toggled on and in the dropdown menu set the distribution group, if any has been created, or a store. If the distribution is toggled in Build configuration menu, upon finishing the build, the binary will be sent to the distribution group or the store, depending on which was selected.

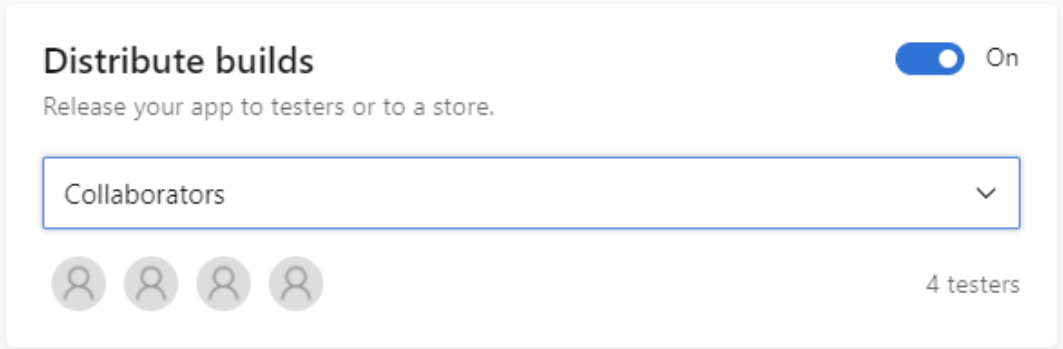

Figure 5. Build distribution in Build configuration menu

If in distribution was selected a distribution group, then the distribution is completed, and the testers or possible users have received their new version of the application in the email.

As it was mentioned before, it is possible also to send the binary to the digital distribution services, in this case, Google Play, as well. If the project had not been connected to a store yet, it is possible to click under stores “Connect to Stores…”. Clicking that option redirects the user to stores tab under distribute menu. For Android, it is possible to connect to Google Play or Intune Company Portal. While Google Play is only for Android applications, Intune Company Portal accepts packages from all platforms. However, due to the requirements from clients, it is only necessary to connect to Google Play.

Connection App Center to Google Play requires a security token from Google Dev Console in a form of a .json file. This token authenticates the user and grants access to publishing the application. To obtain this .json file, first, it is necessary to go to Google Play Console.

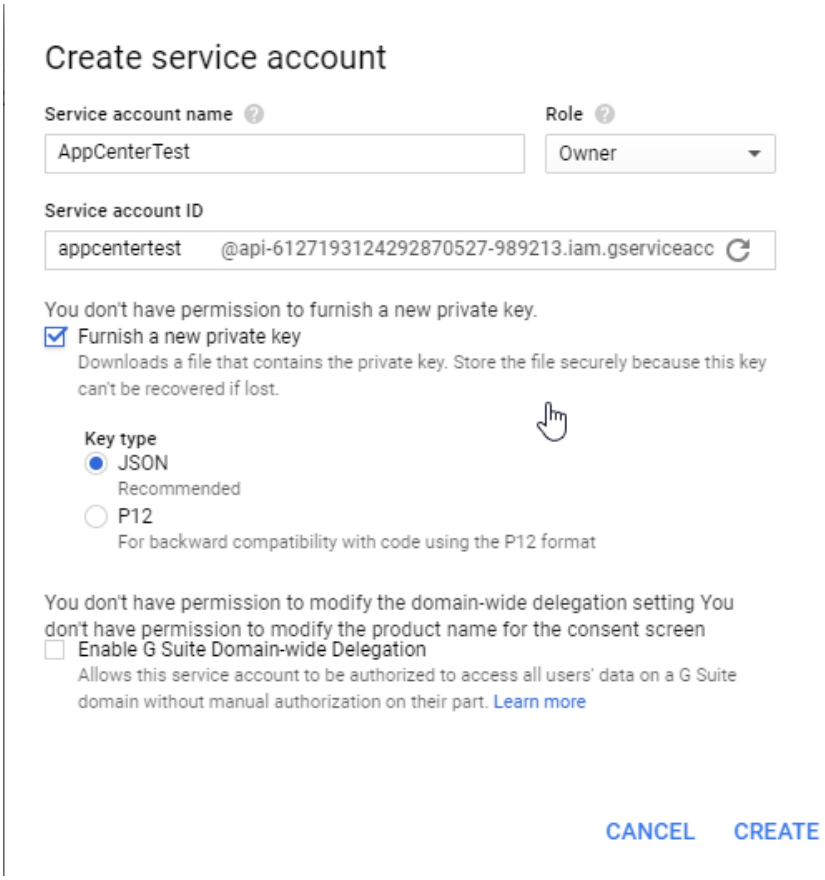

Figure 6. Create service account menu in google developers console

Once the user is logged in, under settings there is API access. In API access menu there are settings related to linked projects, OAuth Clients and Service Accounts. From that, it is necessary to only alter Service Accounts by pressing “create service account”. When the button is clicked, a pop up appears notifying that it is necessary to move to Google API Console via provided link, where it is necessary to create a service account. (Brinke & pnghub 2017.)

In Google API Console there is a “create service account” button in the upper bar which is supposed to be clicked. Once clicked a pop up appears as seen in figure 6, where it is necessary to write the name of the service and which role does the service have. Under those text fields, there is a service account ID, which is generated using the name of service account and a generated key. Under the service account ID, there is a checkbox which needs to be tapped and then it is possible to choose the .json file, which App Center requires. After clicking create, a .json file is downloaded and service account is created. Because this key cannot be recovered, it is important to back up this file and store it in a safe place. Lastly, back in API access page, the service account must be granted access by pressing the grant access button and then .json file can be uploaded to App Center. Once this step is completed, it is possible to publish applications to Google Play. (Brinke & pnghub 2017.)

For Android applications, it is possible to distribute the application to Production, Beta and Alpha tracks. Currently, the only one in use will be Production which is selected by default. Because no releases have been made yet, this menu doesn’t show any releases. However, in the top right corner, there is Publish to Store button. By pressing it, a user is prompted to upload an APK. Currently App Center cannot take successful builds right from build section, so the APK needs to be downloaded manually and uploaded here. After that user can write notes for the update, which will be shown on Google Play and lastly a review before sending it to store.

4.2.2 iOS

On iOS side the signing process requires two files, a .mobileprovision file, and a .p12 file. Both of those files can be obtained on Apple Developer pages. The first one is a mobile provisioning profile file.

A mobile provisioning profile can be used either for applications that are for development uses or for distribution. A provisioning profile is used to install the applications on physical devices. When creating a provisioning profile, it needs to be tied to a specific app ID that

needs to already exist. Because in App Center the point is to distribute the application, the mobile provisioning profile needs to be created for distribution. If the provisioning profile exists and hasn't expired, it is possible to just click on it and download it. However, if it is not possible to download the provisioning profile, by clicking the add button.

The process of creating a provisioning profile consists of four steps. First, the user needs to choose what is the provisioning profile for, development or distribution. Here, it is possible to choose is the application is meant for testers, who will receive the application outside of App Store, by selecting the Ad-Hoc type or if it needs to go to App Store. After the type has been chosen, it is necessary to choose which app ID will the provisioning profile be tied to. Once the ID has been chosen, next it is necessary to select the certificate, to which the provisioning profile will be connected to. Then the provisioning profile needs to be named and it is generated.

The second file for signing is a .p12 file, which is a certificate that was used in creating a provisioning profile. The creation of .p12 file requires the user to have a device with a Mac OS. Unlike provisioning profile, the .p12 file needs to be first exported from a .cer file, which can be downloaded from Apple Developers page. However, if the user doesn't have a file, it first needs to be created. To create the certificate, first, the user needs to select the type, like provisioning profile. However, after the type has been selected, the user is prompted to create a Certificate Signing Request (CSR) file on a Mac device. The process of creating the CSR file is described on the webpage. After the CSR file has been

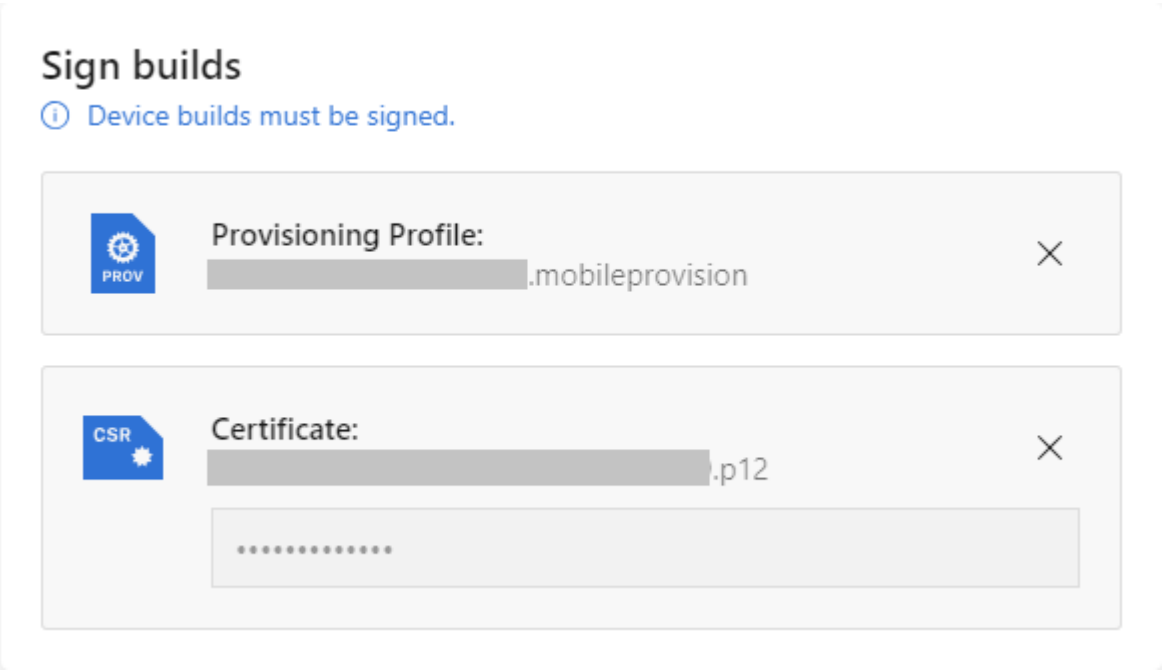

Figure 7. iOS build signing in Build configuration menu

generated, the file needs to be uploaded to the webpage. After pressing the continue button, it is possible to download the .cer file. As mentioned earlier, to get the .p12 file it is necessary to export it from the .cer file and for this, it is preferable to have a Mac device. When .cer file is downloaded to Mac, the file needs to be opened. The certificate will be added to the user keychain and from there it will be possible to export the .p12 file and give it a password.

When both files have been downloaded, now it is possible to put them to App Center to sign the builds. As seen in figure 7, provisioning profile is just uploaded and doesn't require any passwords, because it is linked to the certificate. Certificate, however, has a password, which is configured during the exporting of .p12 file.

Like Android, it is now possible to send this application via emails to testers, if the receiving iPhones are included in provisioning file that was selected to be of type Ad-Hoc.

To distribute the iOS application to App Store, it is necessary to connect App Store to App Center. Under distribute menu in stores tab, after clicking the connect to store button, the user must select App Store. Then the user must create a new account and enter the apple developer account. The account that is linked to App Center can't have a two-factor authentication, because currently, App Center doesn't support it. If the account has successfully logged in, the account appears in the list above the button which adds accounts. By clicking the account which was just added, it is being authenticated and then App Center prompts the user to select, which application should the project associated with. For Duodecim.iOS project, the Duodecim app should be chosen. Now the application can be published to App Store like how Android applications are published to Google Play.

After the application has been sent from App Center to App Store, Apple still reviews that application, possibly declining it. When an application is sent, the status of the publish changes to Submitted, to indicate that the application was sent for review. When the application is published, the status of the app will change to Published. (pnghub 2017.)

4.3 Results

When build and distribution processes were ready, the deployment pipeline was completed. In the test runs that were done, in 15 minutes the build could automatically go from accepted pull request to a functional binary in the email without any assistance from a developer. However, installing the application was successful only on Android. Installing applications on iOS devices that are not directly from App Store requires creating an Ad-Hoc

provisioning profile, which would allow only specific phones to install the application. In addition, it is possible to install the application through TestFlight, however, that was not tested or implemented.

Distribution to the stores in App Center requires one version of the application to already exist on the store. Due to this reason, the application that was supposed to test the App Center distribution to the stores was uploaded first time manually. Because of this, the distribution to stores has not yet been tested. While it is planned to test the distribution in near future, it will not be part of this project anymore.

5 Additional CI configurations

After the deployment pipeline was completed, it was required to also create additional configurations, which would allow developers to upkeep good practices and react quickly to problems. This chapter will cover configurations which were added after the pipeline was completed.

5.1 Bug tracker

In App Center, it is possible to collect diagnostics of the application once it has been released. Using bug tracker, it is possible to automatically create issues in the repository service. As seen in figure 8, it is possible to set the number of the crashes that will trigger the creation of an issue.

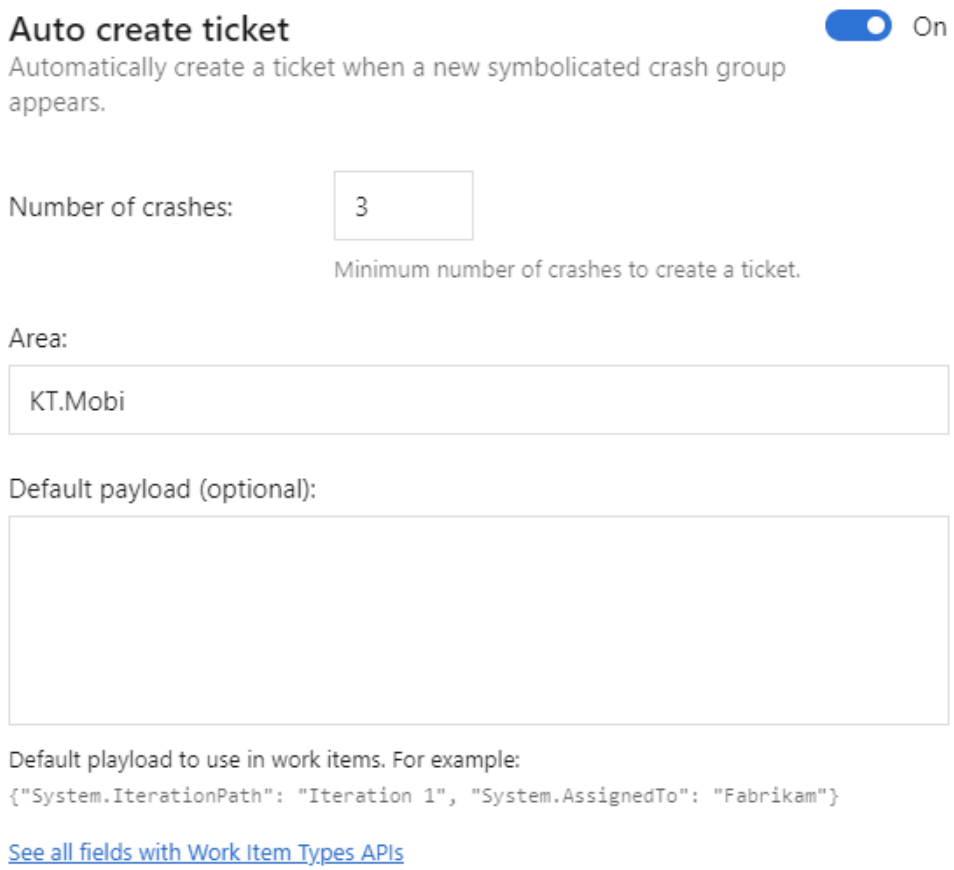

Figure 8. Bug tracker configuration menu

While bug tracker is already implemented as a service to App Center, it hasn't yet been tested. This is because none of the applications have yet been published via App Center, neither do the applications have the necessary NuGet packages supplied by App Center, to support the sending of crash reports from the applications.

5.2 UI tests

While tests are an integral part of continuous integration (Amazon Web Services(AWS).), creating a full testing suite was out of the scope of this project. However, in App Center in Build Configuration menu, it is possible to toggle a switch called Test on a real device, as seen on figure 9, to let App Center run a launch test as soon as the application is built.

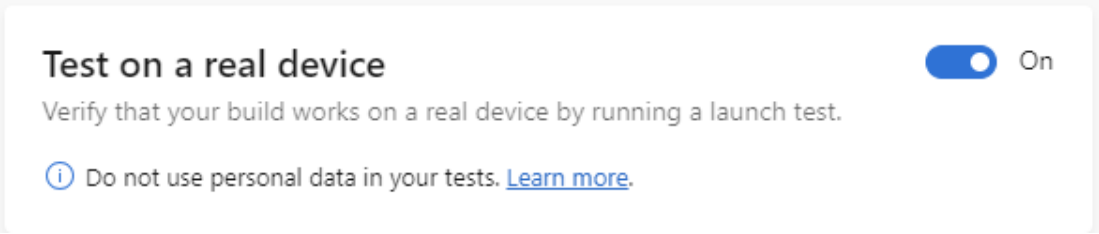

Figure 9. Toggle for launch test in Build Configuration menu

This UI test checks if the application starts and after App Center has run the test it creates a log in the Test section of App Center. In this log, it is possible to see screenshots of the application at each of the steps that were done during testing, stats about the phone and which phones were used during the test.

5.3 Webhooks

In addition to webhooks that App Center creates to repository service, it is possible to create webhooks in projects in App Center. Currently, App Center has four triggers for webhooks. Two of them are related to build, whether it was successful or not, one is if the new version has been released and last one if there is a new crash group created via the bug tracker. To create a webhook in App Center it is necessary to give it a name, write the URL, which the HTTP POST payload will be sent to and choose at least one of the triggers. (Chew 2018.)

To aid mobile team to see when a build has failed, it was decided to create a webhook that would be triggered, if the build that was previously successful after building failed. It was also necessary to see details such as the name of the project and which build failed. Because main communication medium for developers is Slack, it was decided to create a separate channel on Slack that would be receiving all webhook notifications.

To allow Slack to accept incoming webhooks first it was needed to set up the incoming webhooks through settings in Slack. In Slack App Directory under Custom Integrations tab, there is a section which is called Incoming WebHooks. There, it is possible to add configurations. When adding a new configuration, the first thing that needs to be set is the

channel which will receive the webhooks. After that is done, Slack automatically generates the Webhook URL, which then needs to be put in App Center URL text field. (Slack.)

As seen in figure 10, the information which was required is provided. Also, by creating this kind of webhook, it was possible to inform the developers about a broken build without filling the slack channel with notifications of failed build during the possible attempts to fix it.

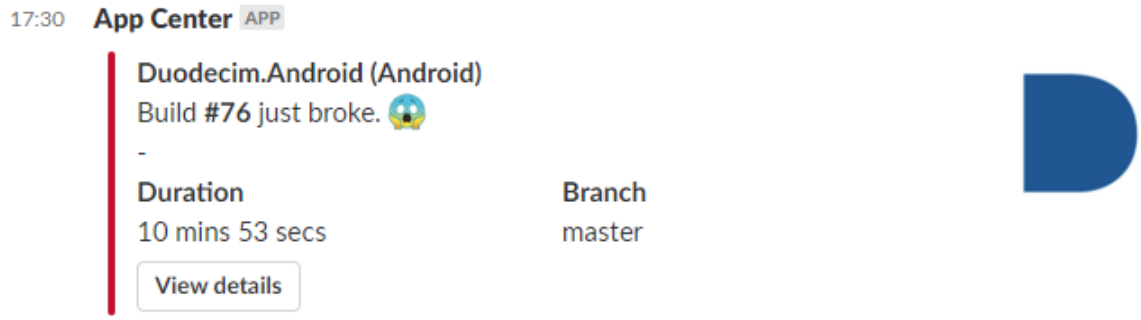

Figure 10. Message from App Center in Slack

5.4 Build status badge

In build configurations, there was a section called Advanced. Under advanced settings, there is a switch for a function called Build status badge. Once toggled, it provides either an image URL or a markdown text containing the link, that could be included in Readme file in Git. This build status badge can display if the corresponding build was successful, failed or it was never built.

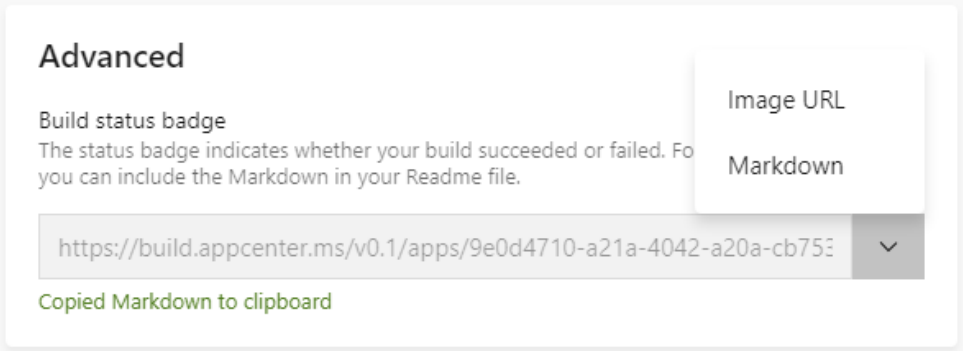

Figure 11. Build status badge in Build configuration menu

Due to the big number of projects in Kehätieto, for convenience, it was decided to take build status badges from all projects and put them in the on the main page of documentation for mobile development. Those status badges were put in a table as seen in figure 11. Since only master branches are configured in a way that they build on push, status badges are taken only from master branch.

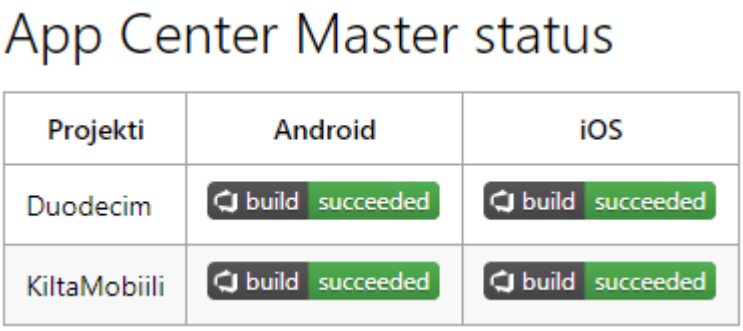

Figure 12. Status badge table from documentation

6 Conclusion

The goal of the project was to create a deployment pipeline from start to finish. When the project was started, it was clear that every part of the integration had to be done from the ground up. However, the goal was achieved. The build process can successfully build the code from the repository and depending on the branch that it takes the code from configure it differently for different purposes. The deployment process, while not fully tested yet, is mostly done. Lastly, the additional CI configurations from App Center are showing their usefulness already and are successfully working without trouble.

6.1 Future of App Center in Kehätieto Oy

App Center was primarily created to be a deployment pipeline for the application with possibility to test the application and distribute them to testers. Due to the scope of the project, functionalities such as diagnostics, analytics, and push notifications weren't worked on. When it was discussed if there would possibly be a need for further development, the analytics and push notifications were declined right away.

While having everything on one platform would be convenient, the mobile team felt that there is no need to collect analytics. Reason for this was that everything necessary is already being collected by the digital distribution services and even then, the information is rarely looked at.

With push notifications, mobile team is attempting to simplify the process from what it currently is and doesn't want to have any more layers than there already is.

The only feature that mobile team was potentially interested in was diagnostics, which includes errors and crashes. Diagnostics in App Center can collect caught and uncaught crashes and when enough of some crash happens, it can create an issue in version control service using a webhook. However, it was decided that this would be something added on-demand.

Currently, no further development is in plans, however, it is likely that the publishing and testing processes will be improved and worked on.

References

Adams, D. 27 June 2017. Build scripts. Visual Studio App Center docs. URL: https://docs.microsoft.com/en-gb/appcenter/build/custom/scripts/. Accessed: 7 April 2018.

Adams, D. 11 June 2017. Environment variables. Visual Studio App Center docs. URL: https://docs.microsoft.com/en-gb/appcenter/build/custom/variables/. Accessed: 3 May 2018.

Amazon Web Services(AWS). What is Continuous Integration? URL: https://aws.amazon.com/devops/continuous-integration/. Accessed: 29 April 2018.

Android Developers. Sign Your App. URL: https://developer.android.com/studio/publish/app-signing. Accessed: 13 May 2018.

Ballinger, K. 15 November 2017. Introducing App Center: Build, Test, Distribute and Monitor Apps in the Cloud. Visual Studio App Center Blog. URL: https://blogs.msdn.microsoft.com/vsappcenter/introducing-visual-studio-app-center/. Accessed: 3 March 2018.

Brinke, S & pnghub, 28 September 2017. Google Play Store Distribution. Visual Studio App Center Docs. URL: https://docs.microsoft.com/en-us/appcenter/distribution/stores/googleplay. Accessed: 13 May 2018.

Chadwick, R. 2017. What is a Bash Script? Ryan's Tutorials. URL: https://ryanstutorials.net/bash-scripting-tutorial/bash-script.php. Accessed: 3 March 2018.

Chew, A. 30 April 2018. App Center Webhooks. Visual Studio App Center docs. URL: https://docs.microsoft.com/en-gb/appcenter/dashboard/webhooks/. Accessed: 10 May 2018.

Collins-Sussman, B. & Fitzpatrick, B.W. & Pilato, C.M. 2011. Version Control with Subversion. URL http://svnbook.red-bean.com/en/1.7/svn.basic.version-control-basics.html. Accessed: 1 May 2018.

Friedman, N. 16 November 2016. Introducing Visual Studio Mobile Center (Preview). The Visual Studio Blog. URL: https://blogs.msdn.microsoft.com/visualstudio/2016/11/16/visual-studio-mobile-center/. Accessed: 3 March 2018.

Ghildiyal, S. & Chandra, P. & Guru99. GUI Testing: Complete Guide. Guru99. URL: https://www.guru99.com/gui-testing.html. Accessed: 15 April 2018.

GitHub. About pull requests. URL: https://help.github.com/articles/about-pull-requests/. Accessed: 3 May 2018.

Goldberger, J. 13 June 2017. Demystifying Build Configurations. Xamarin Blog. URL: https://blog.xamarin.com/demystifying-build-configurations/. Accessed: 13 May 2018.

goldilocks 8 May 2014. What is Binary package? How to build them? StackExchange: Unix & Linux https://unix.stackexchange.com/questions/128462/what-is-binary-package-how-to-build-them. Accessed: 3 March 2018.

KeithS. 22 February 2011. “Debug only” code that should run only when “turned on”. Stack Overflow. URL: https://stackoverflow.com/questions/5080477/debug-only-code-that-should-run-only-when-turned-on. Accessed: 13 May 2018.

Microsoft. Dynamic-Link Libraries. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx. Accessed: 27 March 2018.

Mono. Home. URL: https://www.mono-project.com/. Accessed: 3 May 2018.

Nagele, C. An introduction to version control. Beanstalk Guides. URL: http://guides.beanstalkapp.com/version-control/intro-to-version-control.html. Accessed: 3 May 2018.

Ollari, I. 22 March 2018. Ex-Mobile team project leader. Kehätieto Oy. Email.

Pasat, S. 4 December 2017. Build FAQ. Visual Studio App Center docs. URL: https://docs.microsoft.com/en-gb/appcenter/build/faq#is-my-source-code-secure. Accessed: 15 April 2018.

Pasat, S. & Chew, A. 16 April 2018. Building Xamarin apps for iOS. Visual Studio App Center docs. URL: https://docs.microsoft.com/en-gb/appcenter/build/xamarin/ios/#33-mono-version. Accessed: 3 May 2018.

Pasat, S & Dmitry & Galunin, E & Adams, D 10 May 2018. Cloud Build Machines. Visual Studio App Center docs. URL: https://docs.microsoft.com/en-us/appcenter/build/software. Accessed: 10 May 2018.

Petzold, C. 2016. Creating Mobile Apps with Xamarin.Forms. Microsoft Press. Washington.

pnghub. 31 October 2017. App Store and TestFlight Distribution. Visual Studio App Center docs. URL: https://docs.microsoft.com/en-gb/appcenter/distribution/stores/apple. Accessed: 15 May 2018.

Quinlan, N. 24 June 2014. What's a Webhook? SendGrid. URL: https://sendgrid.com/blog/whats-webhook/. Accessed: 3 May 2018.

Slack. Incoming WebHooks for Slack. URL: https://get.slack.help/hc/en-us/articles/115005265063-Incoming-WebHooks-for-Slack. Accessed: 10 May 2018.

Stone, S. 22 December 2018. Pros and cons of Xamarin Native and Xamarin Forms development—from a native viewpoint. Medium. URL: https://medium.com/@samstone/pros-and-cons-of-xamarin-native-and-xamarin-forms-development-from-a-native-viewpoint-65a06148582. Accessed: 15 April 2018.

ThoughtWorks. Continuous Integration. URL: https://www.thoughtworks.com/continuous-integration. Accessed: 29 April 2018.

Tower. Git Commands. URL: https://www.git-tower.com/learn/git/commands/git-commit. Accessed: 1 May 2018.